SWiFt Developer's Guide

for SWiFt version 1.0

Prepared by: Jerod Wilkerson

Table of Contents

SWiFt Overview

SWiFt is a standard framework for developing web applications. The Enterprise Development Group (eDG) in conjunction with the Web Application Development group of the State of Utah Division of Information Technology Services (ITS) is developing and maintaining SWiFt. Agencies planning to host web applications at ITS are strongly encouraged to use SWiFt as a basis for development. SWiFt provides a common architecture for web applications hosted at ITS and will maximize ITS's ability to provide production support. Agencies not planning to host their web applications at ITS will still receive significant benefits by using SWiFt, and are encouraged to do so.

SWiFt is being developed in an iterative fashion. The first version is available now (in beta), with additional functionality planned for later versions. The current version consists of a Model-View-Controller architecture with support for declarative application security (authorization)[1], application logging, and internationalization. SWiFt supports application security through either the state of Utah's Novell Directory Services (NDS) directory or an XML configuration file. Extensions can also be created to support other security models.

This document describes the SWiFt architecture, with information on how to use it as a basis for web application development projects. It also describes some of the enhancements that are planned for the future.

Design Patterns

SWiFt is based on the following three design patterns:

- Model-View-Controller
- Single-Servlet
- Command

The combination of these three patterns provides a solid foundation for the SWiFt framework based on industry accepted standards and best practices. Each of these design patterns is briefly described below to provide a basis for understanding the SWiFt architecture.

[1] SWiFt's security features are built on an extensible API that can be customized to support any underlying security model. The current version contains API extensions to support Novell Directory Services (NDS) based security using the state's User Master Directory, and simple XML based security using user and role settings specified in an XML file. Additional security extensions can be created to support other security models (such as NT security).

The Model-View-Controller Pattern

SWiFt is based on the Model-View-Controller (MVC) design pattern. MVC separates three distinct forms of functionality within an application. The Model represents the structure of the data in the application, as well as application-specific operations on that data. The View presents data in some form to a user and provides a way for users to interact with the application. An application may contain multiple Views. The Controller handles the interactions between the View(s) and the Model of an application. The Controller eliminates dependencies between the View(s) and the Model.

The MVC design pattern provides several benefits. Separating the Model from the View(s) makes it easy to add additional types of Views for the same data. This separation also allows Model and View components to vary independently, making systems more adaptable, maintainable, and extensible. Having a Controller between the View(s) and the Model eliminates dependencies between the View(s) and the Model. This allows the Model to be changed (for example, the database can be changed from Access to Oracle, or from flat files to a relational database) without affecting the View(s). It also allows the View(s) to be changed without affecting the Model. The Controller permits run-time selection of appropriate Views based on workflow, user preferences, or Model state, and allows application functionality to be reused by different types of Views. The Controller also allows configurable mapping of user actions to application functions. The benefits of the MVC design pattern are so well known and widely accepted that it is the basis for most information systems being developed today.

The Single-Servlet Pattern

The Single-Servlet pattern (an implementation of the Front-Controller pattern) centralizes View management, navigation, and security for a web application in a single object that handles incoming client requests. This single object is a Servlet that acts as the Controller in a Model-View-Controller based web application. The Servlet responds to user requests by translating the requests into application commands and executing them as described in the "The Command Pattern" section of this document.

The Single-Servlet pattern provides the following benefits: 1) facilitates straightforward maintenance of application navigation, 2) facilitates consistent application of security policies across all Views in the system, 3) allows Views to be changed and reused independently of each other, 4) simplifies application deployment because there is only one Servlet to deploy.

The Command Pattern

The command pattern consists of a set of Command objects representing individual units of application functionality, and a Factory object used to determine which Command object will be created and invoked to handle each request. This pattern is often used in conjunction with the Single-Servlet pattern in web applications. The command pattern provides a convenient separation of business logic from other logic. It also separates individual units of business logic from each other. This separation simplifies initial development and enhances adaptability and maintenance of the code.

Struts

Struts is an open-source framework for building web applications that is based on the three design patterns described previously (Model-View-Controller, Single-Servlet, and Command). It is part of the Jakarta project of the Apache Software Foundation. Struts contains much of the functionality required by any web application, so it is used as the core of the SWiFt framework. This section describes Struts in the context of a Model-View-Controller architecture. An understanding of Struts is a prerequisite for using SWiFt. For additional information on Struts, refer to the Struts web site (http://jakarta.apache.org/struts/index.html)[2]. You should also read Appendix I ("Using Struts on iPlanet Application Server") if you plan to deploy Struts or SWiFt applications to iPlanet.

The information in this section is provided to give the necessary background information for understanding SWiFt. Although this section contains information on configuring Struts applications, we do not recommend that you develop your application as a Struts application and then add the SWiFt features later. You will find it much easier to develop your SWiFt applications as SWiFt applications from the start. Developers with extensive Struts experience may want to skip ahead to the "SWiFt Security Extension" section of this document.

The Controller Layer

Most of the functionality provided by Struts is found in the Controller layer. The Controller consists of five main classes or groups of classes. Each of these classes is described below.

The ActionServlet Class

The ActionServlet class is the heart of the Struts framework. It serves the dual purpose of being the Servlet in the Single-Servlet pattern and the Factory in the command pattern as implemented by Struts. The ActionServlet class receives HTML 'get' and 'post' requests and interprets them as requests for application functionality.

After some initial processing, the ActionServlet uses the request URI to determine which application action is being requested. An ActionMapping object represents the requested action. The Servlet then creates an instance of an ActionForm object to represent any parameters received from an HTML form (if the ActionMapping object indicates that the request was initiated by an HTML form). If an ActionForm object is created, it can be validated by calling it's validate(…) method. If the ActionForm is valid, the Servlet determines from the ActionMapping whether the request will be included with or forwarded to another URL. If so, the 'include' or 'forward' is performed and processing

[2] The Struts User Guide (http://jakarta.apache.org/struts/userGuide/index.html) is a great source of information. We recommend that you read the Struts User Guide in its entirety after reading this document.

is complete. Otherwise, the Action object indicated by the ActionMapping is created and executed. The Action object returns an instance of the ActionForward class, which specifies a URL to which the request will be forwarded or redirected to display the results. After the request is forwarded or redirected, processing is complete. Figure 1 provides a graphical representation of this process.

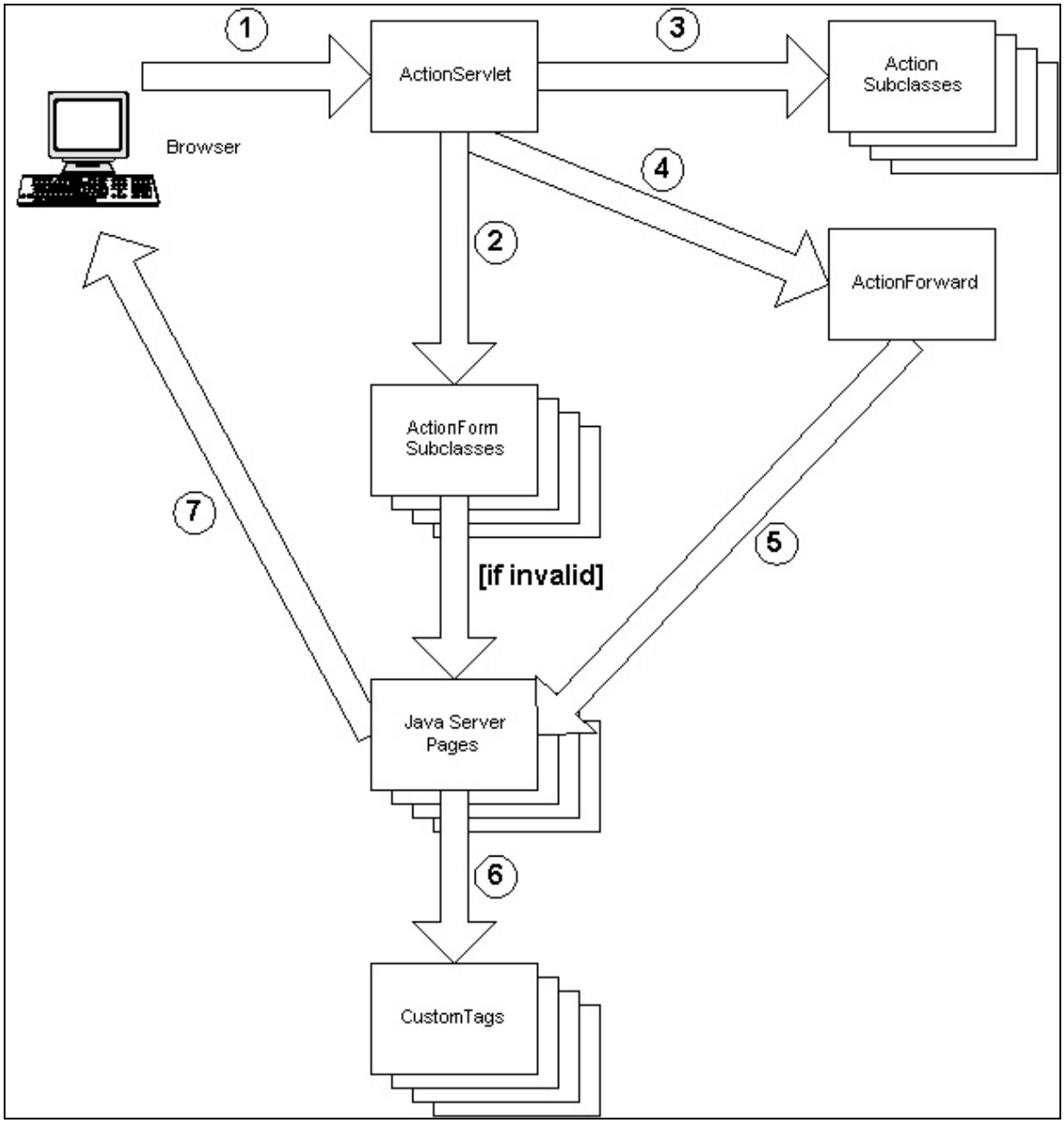

Figure 1: Struts Overview

Additional information about the ActionMapping, ActionForm, Action, and ActionForward classes is provided below.

The ActionMapping Class

Instances of the ActionMapping class represent the configuration information (from the 'struts-config.xml' file) that associates a URI with an action to be performed. The ActionServlet creates the ActionMapping object from the request and uses it to determine how to handle the request. The ActionMapping may specify that a request will be forwarded to or included with another URL. However, most ActionMappings specify an Action that will be performed. If the request was received from an HTML form, the ActionMapping also specifies the class that will be used to represent the data from the form. For additional information on the ActionMapping class, refer to the "Struts Setup and Configuration" section of this document.

ActionForm Classes

ActionForm objects (also known as 'form beans') are used as containers for the data received from 'get' and 'post' requests received from HTML forms. The ActionMapping object indicates which form bean will be instantiated to contain the data. After instantiating the appropriate form bean for a request, the ActionServlet calls mutator (setter) methods on the object to populate it with the request data. If the ActionMapping indicates that the form bean will be validated, the validate(…) method is called. The validate(…) method is inherited from the ActionForm class and by default simply returns null, indicating that the data is valid. Subclasses can override the validate(…) method to validate the data and return an ActionErrors object indicating any errors that were discovered. If an ActionErrors object containing one or more errors is returned, the request is forwarded back to the URL from which it was received, resulting in the HTML form being redisplayed[3]. Struts custom tags in the HTML form provide a simple way to repopulate the form's fields with the data that was submitted and display the error messages on the form. These tags are described in the "Struts Custom Tags" section of this document.

Action Classes

Business logic is added to Struts applications by creating subclasses of the Action class. In a simple application, these Action subclasses may contain the business logic. However, it is generally recommended that Action objects be used to invoke business logic contained in other classes. Action objects contain dependencies on the HTTP protocol. These dependencies are undesirable in business logic because they unnecessarily limit the clients that can access the business logic to web clients. Keeping business logic out of Action objects provides a separation between the Controller and Model layers of the application, and makes the business logic available to any type of client.

The ActionServlet invokes an Action object by calling it's perform(…) method. The perform method of the Action class simply returns null. To add functionality to a Struts application, create subclasses of the Action class that override the perform(…) method to either contain or call the appropriate business logic that should be invoked as the result of

[3] The request is actually forwarded to the URL specified by the 'input' attribute of the corresponding action mapping. This URL should normally reference the form that will submit the request.

an HTTP request. The Action object should create Java Beans to represent any data to be displayed to the user and place them in either the request, the user's session, or the application context to make them available to the application's View layer.

The ActionForward Class

The perform method of an Action object returns an instance of the ActionForward class to specify how the result of a request will be displayed to the user. The ActionForward object indicates a URL (usually to a Java Server Page) to which the request will be forwarded or redirected for the purpose of displaying the result.

The View Layer

The View layer of a Struts application consists of Java Server Pages, Struts custom tags, and optional user defined custom tags. It may also contain user defined Servlets (although this is rare).

Java Server Pages

Java Server Pages (JSPs) are used to create dynamic web pages. They provide a convenient way to combine server-side Java code with HTML. The first time a JSP is invoked it is compiled into a Servlet that outputs the specified HTML. The Struts framework does not actually contain any JSPs. It does, however, provide support for invoking user defined JSPs in a variety of ways.

Most JSPs are invoked from the ActionServlet by either forwarding or redirecting requests to a URL specified by the ActionForward object returned by an Action object. However, they can also be invoked to redisplay an input form when a validation error occurs. JSPs may be invoked from other JSPs by using the Struts 'forward' or 'redirect' custom tags or by using features of the JSP specification directly. Invoking JSPs from other JSPs is discouraged, however, because it allows requests to bypass the Controller layer of the application.

Struts Custom Tags

Struts contains an extensive library of custom tags to simplify the creation of JSPs and to make the features of the Struts framework available to the View layer of Struts applications. Several of the tags provide built-in support for Struts' internationalization features. For more information on Struts' support for internationalization, refer to the "Struts Internationalization Features" section of this document. Some of the more important tags are described below. For additional information on all of the available tags, refer to the Struts API documentation. An article entitled "Using Struts" by Larry Maturo is another useful source of information on Struts custom tags[4].

[4] "Using Struts" by Larry Maturo can be found on the Internet at http://stealthis.athensgroup.com/presentations/Model_Layer_Framework/Struts_Whitepaper.pdfl. It is also available as a link from the resources page of the Struts web site (http://jakarta.apache.org/struts/resources.html).

The HTML Tag

The Struts HTML custom tag can be used in place of the standard <html> tag. It creates a standard <html> tag—setting the 'lang' attribute of the tag to the preferred language specified in the user's browser—and places the locale representing the default language in the user's session. Although most of the other Struts custom tags will work without the Struts HTML custom tag, it is a good practice to use the Struts HTML custom tag in all JSPs that are part of a Struts application. The following example shows how the Struts HTML custom tag is used:

```
<%@ page language="java"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>

<html:html>
<head>
</head>
<body>
</body>
</html:html>
```

The Errors Tag

The Struts Errors custom tag provides a convenient way to display validation and other kinds of errors on a JSP page. The tag searches the user's session for an ActionErrors object, and if found, it displays the error message(s) at the point in the page where the tag is located. The Struts Errors custom tag uses the internationalization features of the framework to display internationalized error messages. For more information on internationalization, refer to the "Struts Internationalization Features" section of this document. By default, all error messages are displayed on the same line. To display the messages on separate lines, include the
 HTML tag at the end of each error message to be displayed. The following example shows how the Struts Errors custom tag is used.

```
<%@ page language="java"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>

<html:html>
<head>
</head>
<body>
   <html:errors/>
</body>
</html:html>
```

The Form Tag

The main purpose of the Struts Form custom tag is to make the appropriate form bean available to any Struts Input custom tags (such as text fields, and selection lists) contained within the Form tag. The input custom tags create input fields that are pre-populated with values from the form bean. This is particularly useful for validation. If a form is submitted with validation errors, the ActionServlet forwards the request back to the form that submitted the request. If the Errors tag is used, a list of validation errors is displayed on the page. If the Form tag and input tags are used, the values that were submitted are repopulated in the fields, selection lists, etc. of the form so the user will see the same page they just submitted with the data they submitted, and a list of errors

displayed. The following example shows how the Form custom tag is used with a set of text input tags to create a form. A complete example would also include a submit button. A more complete example is shown after the "The Message Tag" section of this document.

```
<%@ page language="java"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>

<html:html>
<head>
</head>
<body>
   <html:errors/>
   <html:form action="/EnterAddress" focus="firstName">
      First Name: <html:text property="firstName" size="10"/> <br>
      Middle Name: <html:text property="middleName" size="10"/> <br>
      Last Name: <html:text property="lastName" size="30"/> <br>
      Address: <html:text property="address" size="50"/> <br>
      City: <html:text property="city" size="50"/> <br>
      State: <html:text property="state" size="5"/> <br>
      Zip Code: <html:text property="zipCode" size="12"/>
   </html:form>
</body>
</html:html>
```

All ActionMappings that forward to JSPs that contain the Form tag are required to specify a form bean class.

The Input Tags

Struts has a custom tag for each of the input types that are part of the HTML specification. The input tags generate standard HTML input controls that are pre-populated with the values from the form bean in the user's session. The tags populate the controls by calling the accessor method in the form bean matching the property attribute of the input tag. For example, the Text input tag with a property attribute of "firstName" is populated by calling the form bean's getFirstName() method. For this mechanism to work, the methods of the form bean need to match the naming convention for accessor and mutator (getter and setter) methods for Java Beans.

If the session does not contain a form bean, the input controls are still created, but they are not pre-populated. If the form bean does not contain an appropriate accessor method for an input tag, the input control is not pre-populated.

The Message Tag

The Struts Message custom tag provides support for retrieving internationalized text from locale specific resource bundles and placing it in JSPs. The Message tag uses the language specified in the user's browser to find the appropriate resource bundle from which to retrieve the text. It then uses the key value (from it's key attribute) to retrieve the appropriate internationalized text message from the resource bundle and insert it at the point where the Message tag is specified in the JSP. The following example shows how the Message tag is used:

```
<%@ page language="java"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>

<html:html>
<head>
</head>
<body>
   <html:errors/>
   <html:form action="/EnterAddress" focus="firstName">
      <bean:message key="enter_address.field.first_name"/>
         <html:text property="firstName" size="10"/> <br>
      <bean:message key="enter_address.field.middle_name"/>
         <html:text property="middleName" size="10"/> <br>
      <bean:message key="enter_address.field.last_name"/>
         <html:text property="lastName" size="30"/> <br>
      <bean:message key="enter_address.field.address"/>
         <html:text property="address" size="50"/> <br>
      <bean:message key="enter_address.field.city"/>
         <html:text property="city" size="50"/> <br>
      <bean:message key="enter_address.field.state"/>
         <html:text property="state" size="5"/> <br>
      <bean:message key="enter_address.field.zip_code"/>
         <html:text property="zipCode" size="12"/> <br>
      <html:submit>
         <bean:message key="button.submit"/>
      </html:submit>
   </html:form>
</body>
</html:html>
```

This example assumes that the application contains one or more properly formatted and named resource bundles with values for each of the keys specified in the message tags.

Custom Tag Summary

This has been a very brief overview of some of the custom tags that are part of the Struts framework. Struts contains several other custom tags that have not been described here. Developers can also create their own custom tags for use with Struts applications.

The Model Layer

Struts does not provide direct support for the Model (business logic and data access) layer of a web application. Developers are free to use Enterprise Java Beans (EJBs), Data Access Objects (DAOs) or any other technology or design pattern for developing business logic. In Struts applications, business logic is connected to the Struts framework through the perform(…) method of the Action objects. In a simple application, the Model can be combined with the Controller by placing the business logic in the perform(…) methods of the Action objects. In most applications, the business logic should be separated from Action objects by using either the Data Access Object design pattern or the EJB component model. This provides the benefit of making the business logic available to non-web applications as well. When business logic is separated from Action objects, the Action objects serve the following purposes: 1) calling the appropriate business logic, 2) receiving data to be displayed in the JSPs and placing it in either the

request, the session, or the application context for use by the JSPs and custom tags, 3) forwarding the request to the appropriate JSP after the business logic is executed.

Struts Setup and Configuration

To use Struts as the framework for a web application, you must include the 'struts.jar' file in your application's 'war' file. You will also need to make the 'struts.jar' file available on your CLASSPATH environment variable so you can compile your code. Placing the file on your CLASSPATH does not eliminate the need to place it in the application's 'war' file. DO NOT make the 'struts.jar' file accessible from the CLASSPATH of your Servlet container or application server[5]. The 'struts.jar' file can be downloaded as part of the binary distribution from the Apache Software Foundation's web site (http://jakarta.apache.org/site/binindex.html)[6]. We recommend that you use the latest release build instead of the milestone or nightly builds. The easiest way to get started building a Struts application is to use the blank project included with the binary distribution. To start, simply unzip the 'struts-blank.war' file from the 'webapps' directory, and build your application in the resulting directory structure. For additional information on the directory structure and files required of a Struts application refer to the "Application Configuration and Deployment" section of this document. Information is also available in the Struts User Guide.

To configure a Struts application, edit the 'struts-config.xml' file located in the application's 'WEB-INF' directory. The following sections describe the various tags in the 'struts-config.xml' file with examples of how to modify them. For additional information refer to the Struts User Guide.

Data Sources

The <data-sources> tag allows you to configure connection pools to various data sources for use by your application. You will normally want to use the connection pool features of your Servlet container or application server instead, so we will not describe how to configure and use Struts data sources.

Form Beans

The <form-beans> tag allows you to specify the form bean classes used to represent the data received from your HTML forms. The <form-beans> tag contains one or more <form-bean> tags. A <form-bean> tag simply associates a logical name with a fully qualified package name of a form bean that provides the accessor and mutator methods

[5] The ActionServlet class maintains state information that is specific for the web application it controls. If the ActionServlet class is loaded from the Servlet container's CLASSPATH, multiple Struts applications deployed to the same Servlet container will use the same state information, resulting in unpredictable behavior.

[6] The 'struts.jar' file is also contained in the SWiFt distribution, which is available on the eDG website (http://edg.utah.gov).

for the parameters of an HTML form. The following is an example of a <form-beans> tag containing one <form-bean> tag:

```
<form-beans>
   <form-bean name="enterAddressForm"
      type="gov.utah.swift.swiftexample1.bean.actionform.DisplayAddressForm"/>
</form-beans>
```

The <form-beans> tag contains one <form-bean> tag for each form used in the application. A <form-bean> tag is required for every JSP that uses the Struts Form custom tag. The logical names specified in the <form-bean> tags are used in the <action> tags (described in the "Action Mappings" section of this document) to specify the form bean classes used to receive parameters for specific requests.

Global Forwards

The <global-forwards> tag allows you to specify URLs to which requests can be forwarded. The <global-forwards> tag contains one or more <forward> tags. A <forward> tag associates a logical name with a complete or application relative URL. This logical name can then be used from Action objects or from some of the Struts custom tags to specify a URL to which a request is to be forwarded. The main benefit of associating a logical name with these URLs is that it provides a convenient way to avoid "hard-coding" URLs into your application. The following is an example of a <global-forwards> tag containing one <forward> tag:

```
<global-forwards type="org.apache.struts.action.ActionForward">
   <forward name="defaultPage" path="/WEB-INF/jsp/EnterAddress.jsp"
      redirect="false"/>
</global-forwards>
```

The mapping can then be used from an Action object to specify the JSP to which a request will be forwarded after the business logic is executed. The following example shows how Action objects use global forwards (this code appears at the end of the perform(…) method of an Action object):

```
return mapping.findForward("defaultPage");
```

Global forwards should be specified for pages that are accessed from more than one location in an application (such as an error page). A way to specify forwards that are only accessible from a single request is described in the "Action Mappings" section of this document.

Action Mappings

The <action-mappings> tag and the <action> tag(s) it contains are the most important tags used to configure a Struts application. The <action-mappings> tag may contain multiple <action> tags. The <action> tags are represented in a running Struts application by instances of the ActionMapping class described previously (in the "The ActionMapping Class" section of this document). The main purpose of the <action> tag is to specify the Action class used to process a request, the form bean (if any), and the

URL(s) to which the request can be forwarded after processing. The <action> tag contains several attributes, some of which are required while others are optional. The following is an example of an <action-mappings> tag containing one <action> tag:

```
<action-mappings>
   <action
      path="/EnterAddress"
      type="gov.utah.swift.swiftexample1.action.EnterAddressAction"
      name="enterAddressForm"
      scope="request"
      validate="true"
      input="/WEB-INF/jsp/EnterAddress.jsp">
      <forward name="displayAddress" path="/WEB-INF/jsp/DisplayAddress.jsp"
         redirect="false"/>
   </action>
</action-mappings>
```

The Path Attribute

The 'path' attribute specifies the application context-relative path to the action to be performed. The path is used by the ActionServlet to determine which action mapping will be used for a request. In this example, specifying http://your_application/EnterAddress.do will result in the information contained in the <action> tag of this example being used to handle the request[7].

The Type Attribute

The 'type' attribute specifies the fully qualified name of the Action subclass that will handle the request.

The Name Attribute

The 'name' attribute specifies the logical name of the form bean (as specified in the <form-beans> tag) used as a container for the parameters received with the request.

The Scope Attribute

The 'scope' attribute specifies whether the form bean is stored in the request or the user's session.

The Validate Attribute

The 'validate' attribute specifies whether the form bean will be validated. If set to 'true', the ActionServlet calls the form bean's validate() method before calling the perform(…) method of the action.

The Input Attribute

The 'input' attribute specifies the application context-relative path to which the request will be forwarded if a validation error occurs. This should normally be the path to the JSP that will submit requests to this mapping.

---

[7] Using the '.do' extension is the default way to specify that the ActionServlet should handle a request. For information on changing this default, refer to the "4.5.2 Configure the Action Servlet Mapping" section of the Struts User Guide.

Other Attributes

The <action> tag may contain several other optional attributes. Refer to sections 4.3 and 4.4 of the Struts User Guide for more information. Complete information is available in the description of the 'action' element in the DTD for the 'struts-config.xml' file[8].

The Forward Tag

The <action> tag may contain one or more <forward> tags. These <forward> tags function the same as the <forward> tags contained with the <global-forwards> tag described above, except they are only accessible to the action specified by this mapping.

Validation and Error Handling

Validation of form beans has been described in previous sections and will not be repeated here. In some cases, information required to validate form data will not be available until the business logic is executed. In these cases, the validate() method of the form beans may not serve a useful purpose. To turn form bean validation off, specify 'false' as the value of the 'validate' attribute of the <action> tag for your action mapping.

To invoke the same validation behavior from an action as that provided by the form beans, create an ActionErrors object, populate it with ActionError objects representing the individual error messages, and place the ActionErrors object in the request. The key value to be used when placing the ActionErrors object in the request is the value of the 'ERROR_KEY' field of the Action object. The following example illustrates how to make the ActionErrors object available to the Struts Errors custom tag from an Action object:

```
ActionErrors errors = new ActionErrors();
ActionError error = new ActionError("error message key");

errors.add(ActionErrors.GLOBAL_ERROR, error);
request.setAttribute(Action.ERROR_KEY, errors);
```

After placing the ActionErrors object in the request, the Action object should forward the request back to the page that initiated the request by returning an ActionForward object specifying the calling JSP. This technique can also be used for non-validation errors that should be displayed on the page that submitted the request.

Some errors should not be displayed on the same page that initiated the request. These errors are typically handled by forwarding to an error page. There are two main ways to forward to an error page. One way is to specify a global forward for an error page and then return an ActionForward object specifying the error page from the Action that generates the error. This technique works for expected errors, but it does not handle unexpected exceptions that may be thrown from the ActionServlet or any of the objects whose methods may be called by ActionServlet.

[8] The dtd is the 'struts-config_1_0.dtd' file contained in the 'lib' directory of the Struts binary distribution.

The second way to handle these errors is to specify an error page (using the <error-page> tag) in the 'web.xml' file of the application. This causes the Servlet container to automatically forward unhanded exceptions to the specified page. You may want to use a combination of these two techniques by specifying an error page in the 'web.xml' file and specifying the same page as a global forward.

Struts Internationalization Features

Struts provides built-in support for building internationalized web applications. This built-in support dramatically simplifies the creation of internationalized web applications. Struts provides support for the following internationalization features:

- Resource Bundles
- Browser Specified Locales
- Internationalized Text
- Internationalized Error Messages

Each of these features are described in this section. The combination of these features provides a reasonably complete internationalization framework. For information on Struts internationalization features beyond what is covered in this document, refer to the "3.2 Internationalization" section of the Struts User Guide. Additional information is also available in the "Internationalization" document linked from the Index page of the J2SE documentation.

Resource Bundles

Struts provides direct support for Java's resource bundles to allow text (including error messages) to be displayed in the user's preferred language. For information on how to create resource bundles, refer to the Java API documentation for the ResourceBundle class. The default name for Struts resource bundles is 'ApplicationResources.properties'. Resources bundles should be placed in the 'WEB-INF\classes' directory of your project.

Browser Specified Locales

The Struts framework uses the preferred language specified in the settings of the user's browser to determine the resource bundle from which it should retrieve text. If a resource bundle for the preferred locale does not exist, text is retrieved from the default resource bundle[9].

[9] Although browsers allow users to specify multiple languages in order of preference, Struts ignores all but the first specified language. If a resource bundle does not exist for the preferred language, the default resource bundle is used.

Internationalized Text

Internationalized text is retrieved from resource bundles by using the Struts Message custom tag. For information on how to use this tag, refer to the "The Message Tag" section of this document.

Internationalized Error Messages

ActionError objects represent error messages in Struts applications. ActionError objects support internationalization by containing a key to an internationalized error message instead of an actual message. The Struts Errors custom tag uses the error message key to retrieve the appropriate error message from a resource bundle. This means that any Struts application that makes use of the Errors tag must have it's error messages specified in a resource bundle even if the application only needs to support one language. At first this sounds like a disadvantage of Struts. However, resource bundles are useful even for single language applications because they provide a single point of maintenance for error messages and other text displayed by the application.

Struts Summary

Struts is a powerful framework for building web applications. It is based on the Model-View-Controller design pattern, and it supports the construction of internationalized web applications. The binary distribution of Struts (available at http://jakarta.apache.org/struts) comes with a blank project that contains the default setup and configuration required to build a struts application. The steps required to build a Struts application are listed below (not necessarily in order):

1. Download the Struts binary distribution and place the 'struts.jar' file on your system classpath.
2. Unzip the 'struts-blank.war' file and use the resulting directory structure as the directory structure of your application.
3. Create the JSPs required by your application.
4. Create form beans (ActionForm subclasses) for any HTML forms required by your application.
5. Create Action classes (subclasses of the Action class) as required by your application. As a general rule, you will need one Action class for each request.
6. Create classes to support the business logic accessed from your Action objects.
7. Connect the pieces together by editing the 'struts-config.xml' file from the application's 'WEB-INF' directory.
8. Deploy the application to a J2EE compliant Servlet container or application server.

Software development is never as easy as following an eight-step process. However, the Struts framework makes it much easier that it used to be. The remaining sections of this document describe how Struts fits into the SWiFt framework, how to configure and deploy applications that use the additional parts of the framework, and what enhancements to the framework are planned for the future.

SWiFt Security Extension

SWiFt is being developed as a set of extensions to Struts. These extensions are packaged with a specific version of Struts to provide a framework upon which to build web applications. The main extension included with SWiFt 1.0 is an extensible API for handling application security. This API provides a way to create classes that handle application authorization against any security infrastructure (such as LDAP, NT security, XML configuration files, etc.). The current version of SWiFt contains two implementations of this security API. One of the implementations performs Novell Directory Services (NDS) based security using the state's new User Master Directory (UMD) implementation of NDS. The other implementation of the security API handles security based on the settings of an XML configuration file that specifies roles for application users.

This section describes the SWiFt security extension. It starts with a description of the general security API, and then describes the two security API implementations included with this version of SWiFt. It also describes how additional implementations can be created to allow SWiFt to support additional security models, and concludes with a description of SWiFt custom tags that make security information available to JSPs that are part of a SWiFt application. Except as noted here, configuration of SWiFt applications is the same as Struts configuration.

Security Extension Overview

SWiFt obtains the ID of the current user by calling the getRemoteUser() method on the request object for the requests made by the browser. The remote user can be set in the request by any secure authentication mechanism that requires the user to log in before accessing the application's URL(s)[10]. This authentication occurs before the requests reach the SWiFt framework.

After obtaining the user from the request, SWiFt uses an implementation of the extensible security API to authorize or reject the user's request. The implementation that will be used to perform authorization is specified in the <security-config> tag of the application's 'swift-config.xml' file. The <security-config> tag is described in the "The <security-config> Tag" section of this document.

The specified authorizer contains methods that indicate whether the user is authorized to perform requested actions based on the roles assigned to the user in the underlying

[10] ITS uses SiteMinder for authentication and URL protection. However, you can use SWiFt with any authentication mechanism that sets the remote user in the incoming requests.

SiteMinder authentication can be added at deployment time by requesting SiteMinder protection when requesting deployment by ITS. SiteMinder provides "Single Sign-On" support for the application and sets the remote user in the request so the application can determine the user's identify.

security model. Unauthorized requests result in ServletSecurityExceptions being thrown by SWiFt.

Security API Classes

SWiFt's security API consists of the Authorizer interface and the following four classes: AbstractAuthorizer , SwiftActionServlet, SwiftActionMapping, and ServletSecurityException. The interface and each of these classes are described in this section to provide a basis for understanding how to configure application security. Information on how to handle security exceptions is also included.

The Authorizer Interface

The Authorizer interface provides the API used by SWiFt to authorize user requests. Any class that implements this interface can be used by SWiFt to perform authorization. As a result, SWiFt can easily be extended to support any security model.

The AbstractAuthorizer Class

The AbstractAuthorizer class provides a default implementation for most of the methods that make up the Authorizer interface. This default implementation simplifies the creation of classes to support additional security models. This implementation will be sufficient for most if not all classes that implement the Authorizer interface. Subclasses will still need to implement the getRoles() method, which returns the roles assigned to the current user for the current application from the underlying security model.

The AbstractAuthorizer class compares the roles for the current user with the role(s) required to perform the requested action. If a match is found, the user is authorized to perform the requested action.

The SwiftActionServlet Class

The SwiftActionServlet class is a subclass of the Struts ActionServlet class. In a SWiFt application, the SwiftActionServlet class takes the place of the ActionServlet class. SwiftActionServlet serves as the Controller for SWiFt applications and is the single point of access from the View(s). SwiftActionServlet functions the same as the ActionServlet class with the addition of the functionality provided by the SWiFt extensions to Struts.

With each access of the Servlet, SwiftActionServlet obtains the remote user as described in the "Security Extension Overview" section of this document[11]. It then uses the appropriate implementer of the Authorizer interface to perform authorization—as specified by the 'authorizer' attribute of the <security-config> tag from the 'swift-

[11] During application development, it is sometimes helpful to be able to test SWiFt applications without having an authentication mechanism in place. The 'user' initialization parameter is provided for this purpose. If the request object's getRemoteUser() method returns null, the Servlet looks for the 'user' initialization parameter. If the parameter is found, its value is used in place of the value returned by getRemoteUser(). This "back door" does not compromise production security because the parameter is ignored when an authentication mechanism sets the remote user in the request.

config.xml' file. The <security-config> tag is described in the "The <security-config> Tag" section of this document. If the authorizer class indicates that the user is authorized, processing of the request continues. Otherwise, SwiftActionServlet throws a ServletSecurityException.

The SwiftActionMapping Class

The SwiftActionMapping class is a subclass of the Struts ActionMapping class. Instances of the SwiftActionMapping class represent the configuration information in a SWiFt application that associates a URI with an action to be performed. This is the same information represented by the ActionMapping class in a Struts application—with the addition of the role(s) required to perform the action. The classes that implement the Authorizer interface compare these roles to the roles assigned to the current user (from the underlying security model) to determine if the user is authorized to perform the requested action. Refer to the "Security Configuration" section of this document for details about how to configure the security information.

The ServletSecurityException Class

The SwiftActionServlet throws ServletSecurityExceptions as a result of any unauthorized attempt to invoke functionality on a SWiFt application. The recommended way to handle these exceptions is to use the <error-page> tag in the application's 'web.xml' file to configure an error page to which exceptions are automatically forwarded by the Servlet.

Security Configuration

SWiFt configuration is the same as Struts configuration with the following four exceptions:

1. The name of the configuration file is 'swift-config.xml' instead of 'struts-config.xml'
2. The doctype is different (it specifies a different DTD from that used by 'struts-config.xml')
3. A required <security-config> tag has been added
4. Optional <security-role> tags have been added

The 'swift-config.xml' file is still placed in the application's 'WEB-INF' directory. The changes to the doctype and the additional security tags are described below.

Doctype

The XML parser uses the 'swift-config_1_0.dtd' file to validate the 'swift-config.xml' file. To specify this DTD in your application's XML configuration file, use the following doctype tag:

```
<!DOCTYPE struts-config PUBLIC
  "-//State of Utah//DTD Swift Configuration 1.0//EN"
  "http://webassets.utah.gov/dtd/swift-config_1_0.dtd">
```

The <security-config> Tag

The <security-config> tag specifies information required to perform application authorization. It specifies the fully qualified name of the class used to perform authorization, a URL that can be used by the authorizer to obtain information required to perform authorization (such as the roles assigned to a user), and the identifier by which the application is known to the underlying security model. The <security-config> tag is required, and it must be the first tag defined inside the <struts-config> tag unless an <error-handling> tag is also specified[12]. The following is an example of how to use the <security-config> tag:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE struts-config PUBLIC
  "-//State of Utah//DTD Swift Configuration 1.0//EN"
  "http://webassets.utah.gov/dtd/swift-config_1_0.dtd">

<struts-config>
  <security-config
    authorizer="gov.utah.swift.security.LDAPAuthorizer"
    providerURL="http://authorizer.utah.gov/NASApp/LDAPAuthorizer"
    applicationID="SwiftExample1"/>
```

The 'authorizer' attribute of the <security-config> tag is required. It specifies the fully qualified class name of the class that will be used to perform authorization. The 'providerURL' attribute specifies a URL that can be used by the Authorizer to obtain information from the underlying security model that is required to perform authorization. The 'applicationID' attribute specifies an identifier by which the application is known in the underlying security model. Some authorizer implementations will require values for all of these attributes, while others may not.

The <security-role> Tag

The <security-role> tag is specified inside an <action> tag in the 'swift-config.xml' file. It is used to specify the role(s) that authorize users to perform the action specified by the enclosing <action> tag. An <action> tag may contain any number of <security-role> tags (including zero). Any user that is authenticated for access to the application may perform actions that do not specify any <security-role> tags. For <action> tags that contain multiple <security-role> tags, the action may be performed by authenticated users who have any of the specified roles. The following is an example of how to use the <security-role> tag:

[12] The <error-handling> tag is described in the "Error Handling Override" section of this document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE struts-config PUBLIC
  "-//State of Utah//DTD Swift Configuration 1.0//EN"
  "http://webassets.utah.gov/dtd/swift-config_1_0.dtd">

<struts-config>
  <security-config
    authorizer="gov.utah.swift.security.LDAPAuthorizer"
    providerURL="http://authorizer.utah.gov/NASApp/LDAPAuthorizer"
    applicationID="SwiftExample1"/>
  <action-mappings>
    <action
      path="/EnterAddress"
      type="gov.utah.swift.swiftexample1.action.EnterAddressAction"
      name="enterAddressForm"
      scope="request"
      validate="true"
      input="/WEB-INF/jsp/EnterAddress.jsp">
      <forward name="displayAddress"
         path="/WEB-INF/jsp/DisplayAddress.jsp" redirect="false"/>
      <security-role name="RoleA"/>
      <security-role name="RoleB"/>
    </action>
  </action-mappings>
</struts-config>
```

In this example, authenticated users with either the "RoleA" or "RoleB" role are authorized to perform the "EnterAddress" action.

The NDS Security Implementation

SWiFt contains an implementation of the Authorizer interface that supports authorization against the state of Utah's NDS directory. The class that implements NDS authorization is LDAPAuthorizer. The LDAPAuthorizer class obtains role information from the state's NDS directory to determine whether users are authorized to perform the requested actions. The LDAPAuthorizer class is the Authorizer that should be used for most SWiFt applications developed by agencies of the State of Utah.

The "The <security-config> Tag" section of this document describes how to specify the Authorizer for SWiFt applications by using the <security-config> tag in the 'swift-config.xml' configuration file. When using the LDAPAuthorizer, the 'providerURL' attribute of the <security-config> tag is required. It specifies the URL to a Servlet the Authorizer uses to obtain role information for the current user and application from the state's NDS directory[13]. The 'applicationID' attribute of the <security-config> tag is also required when using LDAPAuthorizer. It specifies the name by which the application is known in the NDS directory. The following is a sample <security-config> tag for use with LDAPAuthorizer:

[13] The URL for the Servlet is http://authorizer.utah.gov/NASApp/LDAPAuthorizer.

```
<security-config
  authorizer="gov.utah.swift.security.LDAPAuthorizer"
  providerURL="http://authorizer.utah.gov/NASApp/LDAPAuthorizer"
  applicationID="SwiftExample1"/>
```

NDS Security Administration

The NDS security implementation depends on information in the state's NDS directory. The following steps are required to specify the necessary security information in the directory:

- Application ID Creation
- Administrator Assignment
- User Account Creation
- User Profile Creation
- Role Creation
- Role Assignment

A security administrator for the application performs these steps. For detailed information on application security administration when using LDAPAuthorizer, refer to the "SWiFt NDS Administrator's Guide".

The XML Security Implementation

SWiFt contains an implementation of the Authorizer interface that supports authorization against an XML configuration file. The class that implements XML based authorization is XMLAuthorizer. The XMLAuthorizer class obtains role information from an XML file for the current user to determine whether users are authorized to perform the requested actions. The format for the XML configuration file used by XMLAuthorizer is specified by the DTD at: http://webassets.utah.gov/dtd/xml-authorizer_1_0.dtd. The following is an example XMLAuthorizer configuration file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE struts-config PUBLIC
  "-//State of Utah//DTD XML Authorizer Configuration 1.0//EN"
  "http://webassets.utah.gov/dtd/xml-authorizer_1_0.dtd">

<authorizer-config>
  <user id="cn=juser,ou=PUBLIC,o=CZ">
    <role>EnterAddress</role>
  </user>
</authorizer-config>
```

The 'id' attribute of the <user> tag must be the value that will be returned by the getRemoteUser() method of the request object for users that will use the application. The value that is returned by the getRemoteUser() method is determined by the authentication

mechanism that authenticates users who access the application[14]. Each <user> tag can contain multiple <role> tags. The <role> tags specify all of the roles assigned to the user for the current application[15].

When using XMLAuthorizer, the 'providerURL' attribute of the <security-config> tag is required. It specifies the location and file name of the XML configuration file used by XMLAuthorizer. For use with XMLAuthorizer, the value for 'providerURL' must be a relative URL referring to a document in the application's 'WEB-INF' directory. The 'applicationID' attribute of the <security-config> tag is not required when using XMLAuthorizer. The following is a sample <security-config> tag for use with XMLAuthorizer:

```
<security-config
  authorizer="gov.utah.swift.security.XMLAuthorizer"
  providerURL="/WEB-INF/xml-authorizer.xml"/>
```

Accessing the Security API

SwiftActionServlet places the Authorizer object in the user's session during the first access of SwiftActionServlet. This object serves as an API to SWiFt's security model that can be accessed from Java classes and JSPs. This provides programmatic access to the same information used by SWiFt to handle authorization. The Authorizer object provides access to the list of roles of the current user for the current application, and provides a method to determine whether the user is authorized to perform specified actions. Although this direct access will rarely be needed, it is a powerful feature of SWiFt

The key under which the Authorizer object is stored in the user's session is the application ID with the value returned by SwiftActionServlet's AUTHORIZER_SUFFIX static variable appended to the end. The following is an example of how to get the Authorizer object for the "swift-example1" application (assume that the application ID is "SwiftExample1"):

```
Authorizer authorizer = (Authorizer) session.getAttribute("SwiftExample1" +
  SwiftActionServlet.AUTHORIZER_SUFFIX);
```

The application ID is returned in each request that accesses SwiftActionServlet, so you can obtain the Authorizer object without knowing the application ID as long as you have access to a request that was handled by SWiFt. This is useful for accessing the Authorizer from JSPs returned by SwiftActionServlet as shown in the following example:

---

[14] For applications using SiteMinder to authenticate against the state's NDS directory, the value returned by the getRemoteUser() method will be the current user's NDS common name.

[15] The XMLAuthorizer's configuration file applies to only one application, so the "current application" is the application containing the configuration file.

```
HttpSession session = pageContext.getSession();
String applicationID = (String) pageContext.getRequest().getAttribute(
  Authorizer.APPLICATION_ID_KEY);
Authorizer authorizer = (Authorizer) session.getAttribute(applicationID +
  SwiftActionServlet.AUTHORIZER_SUFFIX);
```

Security Custom Tags

Four custom tags (UserHasRole, UserNotHasRole, UserAuthorized, and UserNotAuthorized) are included with SWiFt to provide access to security and role information to the View layer of a SWiFt application without requiring developers to write scriptlets that access the Authorizer directly. These tags provide a convenient way to create dynamic JSPs that vary their content depending on whether the user has specified roles or has security access to specified actions.

The security custom tags depend on the Authorizer object that is stored in the user's session the first time the 'SwiftActionServlet' class is accessed. As a result, they do not have the information necessary to determine if a user has a specified role or is authorized to perform an action until at least one action has been invoked. The tags behave in a pessimistic way when the required information is not available by assuming that the user does not have the specified role or is not authorized to perform the specified task[16].

The UserHasRole Tag

The UserHasRole tag causes the JSP to render any HTML between it's opening and closing tag if the user has the role specified by the required 'role' attribute. If the user does not have the specified role, any HTML between the opening and closing UserHasRole tag is ignored. The following example illustrates how the UserHasRole tag is used:

```
<%@ page language="java"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/swift-security.tld" prefix="security"%>

<html:html>
<head>
</head>
<body>
  <security:userHasRole role="EnterAddress">
    This text is only rendered if the user has RoleA.
  </security:userHasRole>
</body>
</html:html>
```

The UserNotHasRole Tag

The UserNotHasRole tag works like the UserHasRole tag, except it only renders enclosed HTML if the user does not have the specified role.

[16] This is normally only an issue from a default page that is displayed when a user first accesses a SWiFt application without specifying a URL for a SWiFt action. The security custom tags are not useful on these pages because the required security information is not yet available.

The UserAuthorized Tag

The UserAuthorized tag causes the JSP to render any HTML between it's opening and closing tag if the user is authorized to perform the action specified by the required 'path' attribute. If the user is not authorized to perform the specified action, any HTML between the opening and closing UserAuthorized tag is ignored. The following example illustrates how the UserAuthorized tag is used:

```
<%@ page language="java"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/swift-security.tld" prefix="security"%>

<html:html>
<head>
</head>
<body>
  <security:userAuthorized path="/EnterAddress">
    This text is only rendered if the user has security rights
    to access the 'EnterAddress' action.
  </security:userAuthorized>
</body>
</html:html>
```

The UserNotAuthorized Tag

The UserNotAuthorized tag works like the UserAuthorized tag, except it only renders enclosed HTML if the user does not have access to the specified action.

Logging

SWiFt uses Log4J for application logging[17]. Like Struts, Log4J is open-source software that is part of the Jakarta project of the Apache Software Foundation. Log4J is a robust, highly flexible, configurable logging API. This section provides a very brief overview of Log4J and gives information on how to configure it for your SWiFt applications. For detailed information, refer to the Log4J web site (http://jakarta.apache.org/log4j/docs/index.html)[18].

Log4J Overview

Log4J provides a convenient, easy to use, API for logging messages from any application. It uses a configurable logging level to determine which messages will be logged and which will be ignored. One of the best features of Log4J is its ability to log to multiple locations. For example, you can specify that your application should log to a file and to the console at the same time.

Log4J has been integrated with SWiFt (including the Struts component of SWiFt), in a way that allows you to configure Log4J for your application, and have it function the same way from the SWiFt and Struts classes included with your application.

Log4J Setup and Configuration

The 'log4j.jar' file must be included with each SWiFt application, and must not be available from the CLASSPATH of your Servlet container or application server[19]. We recommend that you use the 'log4j.jar' file from the lib directory of the SWiFt distribution (this file is included in the blank project described in the "Using the Blank Project" section of this document).

Log4J can be configured within SWiFt by using either a Java properties file or an XML file. The Java properties file is called 'log4j.properties'. The XML file is called 'log4j.xml'. One of the files must exist in the application's 'WEB-INF' directory for logging to occur. If both files exist in the directory, the XML file is used. For information on how to specify the configuration information in the file, refer to the "Short Introduction to Log4J" document on the Log4J web site. The document provides

[17] We considered using the new J2SE 1.4 logging API instead. However, the J2SE logging API is only available with J2SE 1.4. The current version of most Servlet containers and application servers is J2SE 1.3, so a solution that requires J2SE 1.4 is not reasonable.

[18] The "Short Introduction to Log4J" (http://jakarta.apache.org/log4j/docs/manual.html) is particularly useful.

[19] Making the 'log4j.jar' file available from the Servlet container's CLASSPATH causes all applications to use the same logging configuration.

examples of how to configure Log4J using a properties file, but it does not contain any information on XML configuration[20].

[20] XML configuration is not documented anywhere on the Log4J web site. The best source of information is currently the "InitUsingXMLPropertiesFile" example included with the Log4J distribution or the "swift-example1" example included with SWiFt.

Error Handling Override

Some Servlet containers do not properly forward Errors and Exceptions to error pages as specified in the <error-page> tag(s) of the 'web.xml' file. These containers only partially support the <error-page> tag by restricting the exception types that can be specified to those that are declared to be thrown by the 'service' method of the 'javax.servlet.Servlet' interface. This restriction is not acceptable in cases where all Errors and Exceptions (including unchecked exceptions) should go to the same error page.

SWiFt addresses this issue by providing its own error handling mechanism that overrides the Servlet container's default error handling mechanism and forwards Errors and Exceptions to the appropriate error page as specified in the <error-page> tag(s) of the 'web.xml' file. By default, SWiFt's error handling mechanism is turned off and any unhandled Errors or Exceptions are passed to the Servlet container. To turn SWiFt's error handling mechanism on, specify the <error-handling> tag with an 'override' attribute value of 'true' immediately after the <struts-config> tag in the 'swift-config.xml' file as illustrated in the following example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE struts-config PUBLIC
  "-//State of Utah//DTD Swift Configuration 1.0//EN"
  "http://webassets.utah.gov/dtd/swift-config_1_0.dtd">

<struts-config>

  <error-handling override="true"/>

  <security-config
    authorizer="gov.utah.swift.security.LDAPAuthorizer"
    providerURL="http://authorizer.utah.gov/NASApp/LDAPAuthorizer"
    applicationID="SwiftExample1"/>
  .
  .
  .
</struts-config>
```

With the error handling override feature turned on, SWiFt ensures that Errors and Exceptions thrown from within 'SwiftActionServlet' are forwarded to the appropriate error page regardless of whether the Servlet container properly handles the <error-page> tag[21]. Exceptions or Errors that do not have a corresponding <error-page> tag in the 'web.xml' file are passed to the Servlet container's default error handling mechanism.

[21] This feature only affects Errors and Exceptions thrown from 'SwiftActionServlet'. Any other Servlets will continue to use the Servlet container's default error handling mechanism. However, most applications should not have any other Servlets.

Application Configuration and Deployment

This section describes how to configure and deploy SWiFt applications, including Servlet container setup. It does not describe how to specify security information for your application in the state's NDS directory. For information on specifying application security information in the NDS directory, refer to the "SWiFt NDS Administrator's Guide". Some of the information contained in this section is also specified in other sections of this document. This section brings all of the configuration information together in one section for convenience in showing how it all fits together. The 'swift-example1' sample application shows a completed configuration of a simple SWiFt application. You should also read Appendix I ("Using Struts on iPlanet Application Server") if you plan to deploy SWiFt or Struts applications to iPlanet.

Servlet Container Setup

To run SWiFt, you must have J2SE version 1.3 or later and a Servlet container (or application server) that supports the Servlet API specification version 2.2 or later and the Java Server Pages specification version 1.1 or later. Additional prerequisites are required for building the Struts part of SWiFt from source[22].

Using the Blank Project

The easiest way to configure a SWiFt application is to start with the blank project from the SWiFt distribution and use it as a basis for your application. This section describes how to build an application from the blank project and specifies the changes that need to be made to the various configuration files included in the blank project.

To build an application from the blank project, copy the 'blank' directory from your SWiFt distribution to the desired location of your project and rename the 'blank' directory to the name of your application. The blank project contains the following file and directory structure:

[22] For additional information, refer to the "Prerequisite Software" section of the Struts Installation document on the Struts website (http://jakarta.apache.org/struts/userGuide/installation.html).

```
swift-blank
  |
  +--WEB-INF
  |     |
  |     +--classes
  |     |     |
  |     |     +--ApplicationResources.properties
  |     |
  |     +--lib
  |     |     |
  |     |     +--log4j.jar
  |     |     |
  |     |     +--struts-patched.jar
  |     |     |
  |     |     +--swift.jar
  |     |     |
  |     |     +--UserInfo.jar
  |     |
  |     +--log4j.properties
  |     |
  |     +--log4j.xml
  |     |
  |     +--struts-bean.tld
  |     |
  |     +--struts-form.tld
  |     |
  |     +--struts-html.tld
  |     |
  |     +--struts-logic.tld
  |     |
  |     +--struts-template.tld
  |     |
  |     +--swift-config.xml
  |     |
  |     +--swift-security.tld
  |     |
  |     +--web.xml
  |     |
  |     +--xml-authorizer.xml
  |
  +--index.jsp
```

The ‘classes’ directory is where the compiled class files for your application are placed. It also contains the ‘ApplicationResources.properties’ file. The ‘ApplicationResources.properties’ file contains the localized text messages for the application. This is the default resource bundle that will be used if a resource bundle for the locale specified in the user’s browser does not exist. The messages in the resource bundle will need to be replaced with the text to be displayed in your application. You can add additional resource bundles with translated text to make your application support additional languages and geographic regions. For example, a resource bundle named

'ApplicationResources_es.properties' will be used for applications displayed in a browser where the user has specified 'Spanish' as the preferred language[23].

The 'lib' directory contains the four 'jar' files required by SWiFt applications (log4j.jar, struts-patched.jar, swift.jar, and UserInfo.jar). The 'struts-patched.jar' file contains a patch required to compensate for a bug in iPlanet 6.5 that prevents it from being able to run Struts based applications[24].

The 'WEB-INF' directory contains six tag library descriptors ('tld' files) and the configuration files required by SWiFt applications. Five of the 'tld' files are the tag library descriptors for the Struts custom tags. The other 'tld' file (swift-security.tld) is the tag library descriptor for the SWiFt custom tags. The configuration files are described in separate sections below.

The 'index.jsp' file is a simple Java Server Page that displays text from the resource bundle. This JSP should be replaced with the default page for your application. The 'index.jsp' page is the default page for the application, and because of its location, it is displayed without accessing the SWiFt framework. As a result SWiFt does not secure access to this page[25]. To ensure that all application functionality is secure, we recommend that the 'index.jsp' page be used simply to forward the request to a SWiFt action that is secured by SWiFt. The following example (from the 'swift-example1' sample application) illustrates how the 'index.jsp' page can be used to forward requests to SWiFt actions that are secured by SWiFt:

```
<%@ page import="javax.servlet.jsp.PageContext" %>
<%@ page language="java" %>

<html>
<head>
</head>
<body>
  <% pageContext.forward("/DisplayEnterAddress.do"); %>
</body>
</html>
```

This page provides a window into the application without exposing any unsecured functionality.

[23] If the user has specified a specific dialect of the preferred language or a geographic region for the preferred language, the SWiFt framework will search for a resource bundle for that dialect or region. For example, if the user specifies 'es_MX' as the preferred language, SWiFt will search for a resource bundle named 'ApplicationResources_es_MX.properties'. If not found, it will search for a resource bundle named 'ApplicationResources_es.properties'. If it doesn't find either resource bundle it will use the default 'ApplictionResources.properties' bundle.

[24] For details, refer to Appendix I.

[25] If an authentication mechanism (such as SiteMinder) is in place, it will still protect the page by requiring the user to log in.

The 'web.xml' File

The 'web.xml' file is the standard J2EE application descriptor required for all J2EE applications. It specifies the name of the application, the controller Servlet (SwiftActionServlet) with initialization parameters, the mapping that determines which URLs invoke the controller Servlet, the name of the default page, and the tag library descriptors for the SWiFt and Struts tag libraries used by the application. Most entries in the 'web.xml' file will not need to be changed for your application. This section describes the changes that will need to be made. For additional information not described here, refer to the Struts User Guide.

Application deployment tools use the <display-name> tag to display the name of the application. The text inside this tag should be changed to the name of your application.

The value of the 'user' initialization parameter is used to identify the user when an authentication mechanism is not available. This is sometimes useful in development environments. If your development environment has access to an authentication mechanism that sets the remote user in incoming requests, you can remove the parameter. Otherwise, change it to the identifier of the person who should be considered the user in your development environment.

The <servlet-mapping> specifies which URLs will invoke the controller Servlet. The default setting invokes the controller Servlet for any URL to the application that ends in '.do'. This can easily be changed by changing the <url-pattern> tag inside the <servlet-mapping> tag.

The tag library descriptors included in the 'web.xml' file are a subset of the available tag libraries. Change the set of tag library descriptors to include only those used by your application.

The 'swift-config.xml' File

The 'swift-config.xml' file contains the configuration information for the Struts functionality and the SWiFt extensions. Details of the 'swift-config.xml' file are specified in the "Security Configuration", "Struts Setup and Configuration", and "Error Handling Override" sections of this document. Follow the instructions in these sections to configure your application. Depending on the Authorizer you plan to use for your application, you will need to change the attributes of the <security-config> tag[26]. The 'swift-config.xml' file from the blank project contains commented-out examples of how to configure a SWiFt application.

The 'xml-authorizer.xml' File

The 'xml-authorizer.xml' file specifies the application's users and their corresponding roles when the XMLAuthorizer is being used. The file can be deleted if another authorizer is being used for your application. The 'xml-authorizer.xml' file can be

[26] At a minimum, you will need to either change the 'application ID' attribute to the ID of your application or remove it if the Authorizer to be used doe not require and application ID.

renamed as long as the value for the 'providerURL' attribute of the <security-config> tag in the 'swift-config.xml' file is updated to match the new name.

The 'log4j.xml' and 'log4j.properties' Files

The 'log4j.xml' and 'log4j.properties' files are used to configure application logging. One of the files should be deleted (depending on whether you want to configure logging from an XML or a properties file). If both files exist, the XML file is used. By default, the log messages will be logged to a file named 'swift-blank.log' in the Servlet container's 'logs' subdirectory and only messages with a log level of 'warn' or higher will be logged. You will need to verify that the specified file name and path are valid on your server. For information on changing the logging configuration, refer to the "Short Introduction to Log4J" document on the Log4J web site (http://jakarta.apache.org/log4j/docs/manual.html).

Adding Java Server Pages

Java Server Pages can be placed almost anywhere in your application's directory structure. However, any Java Server Pages that are not specified under the 'WEB-INF' directory can be accessed directly from a web browser (without accessing the SWiFt framework). As a result, we recommend that all Java Server Pages (except the default 'index.jsp' page) be placed in a 'jsp' subdirectory of the 'WEB-INF' directory. These JSPs can then be accessed from 'forwards' specified in the 'swift-config.xml' file that use relative paths from the application base directory as follows:

```
<forward name="defaultPage" path="/WEB-INF/jsp/MyJSP.jsp" redirect="false"/>
```

Adding Classes

The "Action", "ActionForm", and other Java classes required by your application are placed in the application's 'WEB-INF/classes' directory in a directory structure matching the package names of the classes.

Application Deployment

To deploy your application, use the 'jar' tool to create a 'war' file for your application. You can then use the deployment tool for your Servlet container or application server to deploy the application. Deployment tools package the 'war' file in an 'ear' file. You can also use the 'jar' tool to create the 'ear' file manually.

Future Enhancements

SWiFt is being developed and maintained by ITS. However, ITS is considering maintaining the project in an "open-source" fashion in the near future. This would provide a way for any developer within state government to participate in the project.

Plans are being made now for enhancements to SWiFt. We anticipate that in the future, direct support will be added for Web Services, EJB development, access to State of Utah employee data through EJBs, and support for the "Data Access Object" design pattern[27]. Enhancements will also be made to the SWiFt security API. We are also in the process of developing a library of utility classes that will be included as a standard part of SWiFt. These utility classes will also be made available for non-SWiFt applications.

[27] Although SWiFt does not provide direct support for these technologies, they can still be used to build SWiFt applications by accessing them from 'Action' objects.

Appendix I – Using Struts on iPlanet Application Server

This appendix describes problems we have encountered with iPlanet application server, and specifies changes required to make Struts and/or SWiFt applications function properly on iPlanet. Although the changes described here are for iPlanet 6.5, they are backward compatible with previous versions of iPlanet and should also be compatible with any other application server. The 'swift-example1' sample application incorporates all of the changes described here. In addition to these changes, we recommend that you only deploy 'ear' files to iPlanet. We have had sporadic problems with applications that are deployed directly from 'war' files.

The iPlanet 6.5 Struts Patch

A patch must be made to Struts to make it function on iPlanet 6.5. This patch is actually a work-around for an iPlanet 6.5 bug. iPlanet 6.5 does not locate top-level, non-public classes that share a source file with a top-level, public Servlet. Supporting these classes is part of the Java language specification, so this is an iPlanet bug. However, it is easily resolved by making a change to Struts.

Struts has an 'AddDataSourceRule' class that shares a source file with the 'ActionServlet' class. The 'AddDataSourceRule' class can be converted into a class that iPlanet 6.5 can locate by making it an inner class of the 'ActionServlet' class. This change has already been made in the 'struts-patched.jar' file included with SWiFt. The 'struts-patched.jar' file should be used in place of 'struts.jar' for any Struts or SWiFt applications deployed to iPlanet. The 'struts-patched.jar' file should also work with any other Servlet container or application server.

The 'ias-web.xml' File

The 'ias-web.xml' file is required by iPlanet Application Server to specify a globally unique identifier (GUID) for all Servlets in a web application. GUIDs may also be specified for JSPs. The iPlanet deployment tool is supposed to generate the required file if it does not already exist in the application's 'war' file. However, some versions of iPlanet (including iPlanet 6.5) do not generate the file correctly in some circumstances. As a result, we recommend creating the file manually and placing it in the 'war' file's 'WEB-INF' directory for any web application that is to be deployed to iPlanet. The following is the 'ias-web.xml' file of the 'swift-examle1' sample application:

```
<!DOCTYPE ias-web-app PUBLIC '-//Sun Microsystems, Inc.//DTD iAS Web
Application 1.0//EN'
'http://developer.iplanet.com/appserver/dtds/IASWebApp_1_0.dtd'>

<ias-web-app>

  <servlet>
    <servlet-name>action</servlet-name>
    <guid>{57AE2EB9-7179-1D45-B61F-080020B97711}</guid>
    <validation-required>false</validation-required>
    <servlet-info>
      <sticky>false</sticky>
      <encrypt>false</encrypt>
      <number-of-singles>10</number-of-singles>
      <disable-reload>false</disable-reload>
    </servlet-info>
  </servlet>

  <session-info>
    <impl>distributed</impl>
    <dsync-type>dsync-distributed</dsync-type>
    <timeout-type>last-access</timeout-type>
    <secure>false</secure>
    <domain></domain>
    <path></path>
    <scope></scope>
  </session-info>

</ias-web-app>
```

The name specified in the <servlet-name> tag must match the name specified in the <servlet-name> tag of the corresponding 'web.xml' file entry. The value specified for the <guid> tag should be generated by invoking the 'kguidgen' tool from the 'bin' directory of your iPlanet installation. For details on the other tags of the 'ias-web.xml' file, refer to the iPlanet documentation.

A <servlet> tag must exist in the 'ias-web.xml' file for each Servlet in your application. JSPs only require <servlet> tags if they have <servlet> tags in the 'web.xml' file.

Error Handling

IPlanet 6.5 (and probably previous versions) has the error-handling problem described in the 'Error Handling Override' section of this document. This problem is easily resolved for SWiFt applications by turning on SWiFt's error handling mechanism as described in the "Error Handling Override" section of this document.